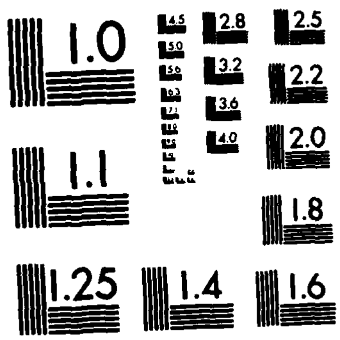
AD-A134 033 COMPUTER PROGRAM DEVELOPMENT SPECIFICATION FOR ADA 1/1
INTEGRATED ENVIRONMENT..(U) INTERMETRICS INC CAMBRIDGE
MA 01 DEC 82 IR-679-1 F30602-80-C-0291

UNCLASSIFIED F/G 9/2 NL


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

AD-A134033

IR-679-1

COMPUTER PROGRAM DEVELOPMENT SPECIFICATION FOR Ada INTEGRATED ENVIRONMENT: MAPSE COMMAND PROCESSOR B5-AIE (1).MCP (1)

1 DECEMBER 1982

CONTRACT F30602-80-C-0291

PREPARED FOR: ROME AIR DEVELOPMENT CENTER
CONTRACTING DIVISION/PKRD
GRIFFISS AFB, N.Y. 13441

PREPARED BY: INTERMETRICS, INC.
733 CONCORD AVE.
CAMBRIDGE, MA 02138

This document was produced under contract F30602-80-C-0291/SA P0009 for the Rome Air Development Center. Mr. Donald Mark is the Program Engineer for the Air Force. Mr. Mike Ryer is the Project Manager for Intermetrics.

TABLE OF CONTENTS

	PAGE
1.0 SCOPE	1
1.1 Identification	1
1.2 Functional Summary	1
2.0 APPLICABLE DOCUMENTS	3
2.1 Program Definition Documents	3
2.2 Inter Subsystem Specifications	3
2.3 Military Specifications and Standards	3
2.4 Miscellaneous Documents	3
3.0 REQUIREMENTS	5
3.1 Introduction	5
3.1.1 General Description	5
3.1.2 Peripheral Equipment Identification	5
3.1.3 Interface Identification	5
3.2 Functional Description	5
3.2.1 Equipment Description	5
3.2.2 Computer Input/Output Utilization	5
3.2.3 Computer Interface Block Diagram	7
3.2.4 Program Interfaces	7
3.2.4.1 KAPSE	7
3.2.4.2 Fundamental Programs	7
3.2.4.3 User Programs	7
3.2.5 Function Description	8
3.3 Detailed Functional Requirements	11
3.3.1 Command Input	11
3.3.2 DRIVER	15
3.3.2.1 Inputs	15
3.3.2.2 Processing	15
3.3.2.3 Outputs	16
3.3.3 LEXPARSE	16
3.3.3.1 Inputs	16
3.3.3.2 Processing	16
3.3.3.3 Outputs	16

INTERMETRICS INCORPORATED • 733 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

TABLE OF CONTENTS (Cont'd.)

	PAGE
3.3.4 ALLOC INTERP	16
3.3.4.1 Inputs	17
3.3.4.2 Processing	17
3.3.4.3 Outputs	17
3.3.5 TREE INTERP	17
3.3.5.1 Inputs	17
3.3.5.2 Processing	17
3.3.5.2.1 I/O Redirection	17
3.3.5.2.2 Pipes	18
3.3.5.2.3 Program and Script Invocation	18
3.3.5.2.4 Other Command Processing	18
3.3.5.3 Outputs	19
3.3.6 BACKGRD MGR	19
3.3.6.1 Inputs	19
3.3.6.2 Processing	20
3.3.6.3 Outputs	20
3.3.7 EXPR PROC	20
3.3.7.1 Inputs	20
3.3.7.2 Processing	20
3.3.7.3 Outputs	20
3.3.8 SCRIPT	21
3.3.8.1 Inputs	21
3.3.8.2 Processing	21
3.3.8.3 Outputs	21
3.3.9 PROGRAM INVOCATION	21
3.3.9.1 Inputs	21
3.3.9.2 Processing	22
3.3.9.3 Outputs	22
3.3.10 VARIABLE	22
3.3.10.1 Input	22
3.3.10.2 Outputs	23

INTERMETRICS INCORPORATED • 733 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

TABLE OF CONTENTS (Cont'd.)

	PAGE
3.3.11 ERROR	23
3.3.11.1 Inputs	23
3.3.11.2 Processing	23
3.3.11.3 Outputs	23
3.4 Adaptation	23
3.5 Capacity	24
4.0 QUALITY ASSURANCE PROVISIONS	25
4.1 Introduction	25
4.2 Test Requirments	25
4.2.1 Unit Testing	25
4.2.2 Integration Testing	25
4.2.3 Functional Testing	26
4.3 Acceptance Test Requirements	26
APPENDICES	
APPENDIX A THE MCL COMMAND LANGUAGE	27
A.1 Help Command	27
A.2 Program Invocation	27
A.2.1 Parameter Passing	27
A.2.2 Function Invocation	30
A.3 Expression Manipulation Commands	30
A.3.1 Assignment	30
A.3.2 GET	31
A.3.3 PUT	31
A.4 Database Commands	32
A.5 Control Commands	32
A.5.1 IF	32
A.5.2 Loop	32
A.6 MCP Termination	33
A.6.1 RETURN	33
A.6.2 LOGOUT	33
A.6.3 SUSPEND	33
A.6.4 Shutdown Commands	34
A.6.5 RESUME	34

TABLE OF CONTENTS (Cont'd.)

	PAGE
A.7 I/O Redirection	34
A.7.1 Standard I/O	34
A.7.2 Pipes	35
A.8 Backgrd Execution of Commands	35
A.9 Compound Commands	37
A.10 MCP Variables	38
A.10.1 MCP Variable Attributes	38
A.11 MCP Predefined Variables	39
A.12 External Program Invocation	39
A.12.1 Context Object Names	39
A.12.3 Control Commands	43
A.13 EXEC	44
A.14 RENAMES	44
A.15 Nested MCP's	44
A.16 Scripts	45
A.16.1 Script Parameters	46
A.16.2 Affecting the MCP's Environment	47
APPENDIX B: MCL LANGUAGE ELEMENTS	49
B.1 Lexical Elements	49
B.2 Types	51
B.3 Expressions	52
B.3.1 Operands	52
B.3.2 Operators	53
B.3.3 Type Conversions	53
B.3.4 Explicit Type Conversions	55
APPENDIX C: BNF FOR MAPSE COMMAND LANGUAGE (MCL)	57
APPENDIX D: STATEMENTS AND FUNDAMENTAL PROGRAMS	65

INTERMETRICS INCORPORATED • 733 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

TABLE OF CONTENTS (Cont'd.)

PAGE

FIGURES AND TABLES

FIGURE 3-1: KAPSE FUNCTIONAL RELATIONSHIP 6
FIGURE 3-2: PARAMETER PASSING 9
FIGURE 3-3: MCP MODULES 10

FIGURE A-1: SAMPLE MCP SESSION 28

TABLE 3.1: COMMAND SUMMARY 13

TABLE A.3: MCP PREDEFINED VARIABLE (%ENVIRONMENT COMPONENTS) 40
TABLE A.4: MCP PREDEFINED VARIABLES (%STATUS COMPONENTS) 40
TABLE A.5: CONTEXT OBJECT ATTRIBUTES 42

TABLE B.1: MCL OPERATORS 54
TABLE B.2: IMPLICIT TYPE CONVERSIONS 55

TABLE D.1: STATEMENTS 66
TABLE D.2: FUNDAMENTAL PROGRAMS 67
TABLE D.3: COMMANDS. ADA - MCL COMPARISON 68
TABLE D.4: LANGUAGE ELEMENTS. ADA - MCL COMPARISION 70

INTERMETRICS INCORPORATED • 733 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

1.0 SCOPE

1.1 Identification

This specification describes the MAPSE Command Language (MCL) with which a user selects AIE facilities, and establishes the requirements for performance, design, test and qualification of the MAPSE Command Processor (MCP), a computer program that interprets and acts upon MCL commands. This specification also identifies interfaces with the KAPSE and with other MAPSE tools that, together, provide the full range of capabilities available to the AIE user.

MCP is classified within the AIE configuration both as a subsystem and as a Computer Program Configuration Item.(CPCI). It consists of the following Computer Program Components (CPC's).

DRIVER(A)
LEXPARSE(B)
ALLOC INTERP(C)
TREE INTERP(D)
BACKGRD MGR(E)
EXPR PROC(F)
SCRIPT PROCESSING(G)
PROGRAM INVOCATION(M)
VARIABLE(I)
ERROR(J)

1.2 Functional Summary

The user communicates with the MCP via an Ada-like interpretive language called MCL (MAPSE Command Language). In response, the MCP provides the following basic capabilities:

(1) invocation of specified MAPSE tools and user-defined programs, and control over their execution;

(2) "help" facility that can be used on a per-program or per-program-parameter basis;

(3) manipulation of built-in MCP variables to obtain status information or to establish command scripts;

(4) redirection of input and output on a per command basis;

(5) the connection of commands as co-routines by specifying the output of one to be the input of the other (via "pipes");

(6) execution of commands in the foreground, background, interactively or in batch mode.

PAGE LEFT BLANK INTENTIONALLY

2.0 APPLICABLE DOCUMENTS

2.1 Program Definition Documents

Requirements for Ada Programming Support Environments, "STONEMAN", February 1980, Department of Defense.

Reference Manual for the Ada Programming Language, draft proposed ANSI standard document, July 1980, Department of Defense.

Revised Statement of Work (15 March 1980).

2.2 Inter Subsystem Specifications

System Specification for Ada Integrated Environment, Type A, AIE(1).

Computer Program Development Specifications for Ada Integrated Environment (Type B5):

Ada Compiler Phases, AIE(1).COMP(1)

KAPSE/Database, AIE(1).KAPSE(1)

MAPSE Generation and Support, AIE(1).MGS(1)

Program Integration Facilities, AIE(1).PIF(1)

MAPSE Debugging Facilities, AIE(1).DBUG(1)

MAPSE Text Editor, AIE(1).TXED(1)

Virtual Memory Methodology, AIE(1).VMM(1).

Technical Report (Interim), IR-684

2.3 Military Specifications and Standards

Data Item Description DIE-30139, USAF, 24 July 1976.

2.4 Miscellaneous Documents

Diana Reference Manual (G.Goos and Wm. A. Wulf, eds.) Institut Fuer Informatik II, Universitaet Karlsruhe and Computer Science Department Carnegie-Mellon University, March 1981.

PREVIOUS PAGE IS BLANK

PAGE LEFT BLANK INTENTIONALLY

3.0 REQUIREMENTS

3.1 Introduction

This section provides the set of requirements for the MAPSE Command Processor (MCP). This includes a brief description of the command language, MCL, with which a user communicates with the MCP. A more detailed language description is provided in Appendices A-C.

3.1.1 General Description

The MAPSE Command Processor serves as a basic interface between the AIE user and the executable programs that reside within it. It accepts as input user commands written in MCL and invokes the appropriate functions to perform the task required. In addition to program execution, the MCP activates KAPSE utility functions and provides its own repertoire of services that permit the user to modify the programming environment (for example, to establish special procedures to be executed automatically at login or to rename MCP commands). An important MCP-specific function is the "help" facility that provides information on using system components on a per-program or per-parameter basis.

3.1.2 Peripheral Equipment Identification

Not applicable.

3.1.3 Interface Identification

The Command Processor interfaces with:

1. the KAPSE;

2. fundamental programs;

3. user programs and scripts.

These interfaces are diagrammed in Figure 3-1, and are described in Section 3.2.4.

3.2 Functional Description

3.2.1 Equipment Description

Not applicable.

3.2.2 Computer Input/Output Utilization

Not applicable.

PREVIOUS PAGE IS BLANK

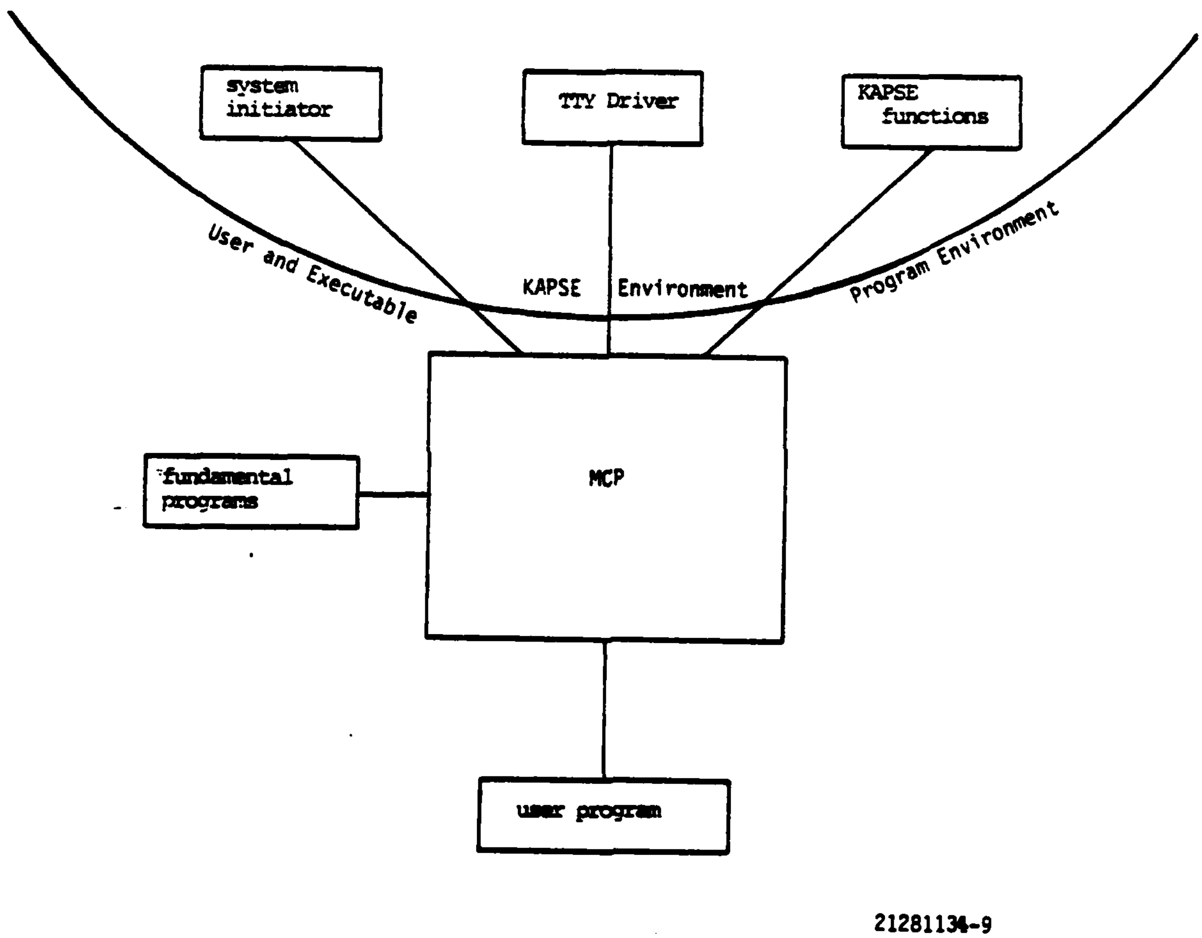


FIGURE 3-1: KAPSE FUNCTIONAL RELATIONSHIP

3.2.3 Computer Interface Block Diagram

Not applicable.

3.2.4 Program Interfaces

3.2.4.1 KAPSE

When a user logs in to the AIE, the system initiator within the KAPSE invokes the MCP on the user's behalf. At the conclusion of the MCP session, control is returned to the system initiator.

The MCP receives its input in interactive mode from the terminal. The terminal driver within the KAPSE provides the user with primitive editing capabilities such as character deletion.

In carrying out user-specified commands, the MCP invokes various KAPSE functions. These are described in detail in Section 3.3.

3.2.4.2 Fundamental Programs

Many MCL commands require manipulation of data structures defined within the MCP (e.g., MCP variables). To change the syntax or effect of such commands would require the recompilation of the MCP.

Commands that do not reference any internal MCP data structures are implemented as linked executable programs called fundamental programs. The syntax or effect of a command implemented via a fundamental program can be modified simply by recompiling the fundamental program.

The MCL language description uses the term "statement" to indicate those commands that are carried out within the MCP. Appendix D lists fundamental programs referenced in this document.

3.2.4.3 User Programs

The MCP user can invoke an arbitrary program and may optionally supply parameters. If the program does not require parameters, the MCP simply makes the KAPSE call INITIATE_PROGRAM. If, however, the program does take parameters, special processing is required on the part of the MCP and of the invoked program. Parameters may include MCP variables supplied as OUT parameters, whose value may be modified as a result of the program's execution.

Each parameterized program must contain a preamble. This preamble processes parameters on behalf of the program's main parameterized subprogram, as follows (see also Figure 3-2).

First, the MCP evaluates the parameters and makes the KAPSE call INITIATE_PROGRAM to begin the execution of the specified program using an MCP-named context object.

The program's preamble reads the user-specified parameter values from the context object. The preamble then calls the program's main parameterized subprogram, supplying it with the user-specified parameter values. When that subprogram has completed, the preamble writes the updated values of any OUT parameters back into the context object. The MCP can then read these updated parameter values from the context object.

A program may also have help information associated with it via "help attributes". These attributes include:

'GENERAL_HELP: The name of a simple object containing general help text for the program. If the program has no general help associated with it, the attribute is undefined.

'PARAMETER_HELP: The name of a composite object containing "parameter help" simple objects. Each parameter help simple object contains help text for a specific program parameter. If no such parameter help text exists, this attribute is undefined.

'OWN_PARAMETER_HELP: If "TRUE", the program interprets the actual parameter values of '?' as a request for parameter help, and supplies this help itself.

3.2.5 Function Description

The MCP is composed of various Ada subprograms, tasks and packages linked together into an executable program. Figure 3-3 shows the flow of control between the components of the MCP. These are described briefly below.

DRIVER, the main program of the MCP, reads user-typed commands, calls LEXPARSE to parse input, and passes a parse tree on to the TREE_INTERP task to be interpreted.

ALLOC INTERP allocates/deallocates a tree interpreter task. TREE INTERP walks and evaluates an input parse tree. BACKGRD MGR handles the background execution of commands. EXPR PROC evaluates MCP expressions (see Appendix B.3).

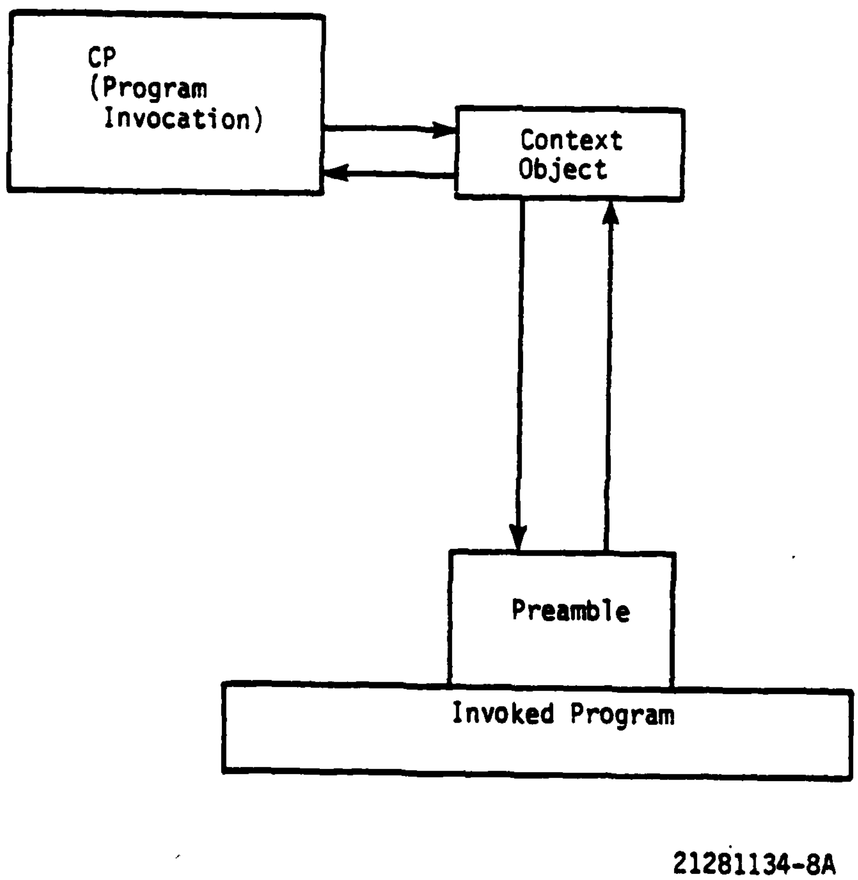


FIGURE 3-2: PARAMETER PASSING

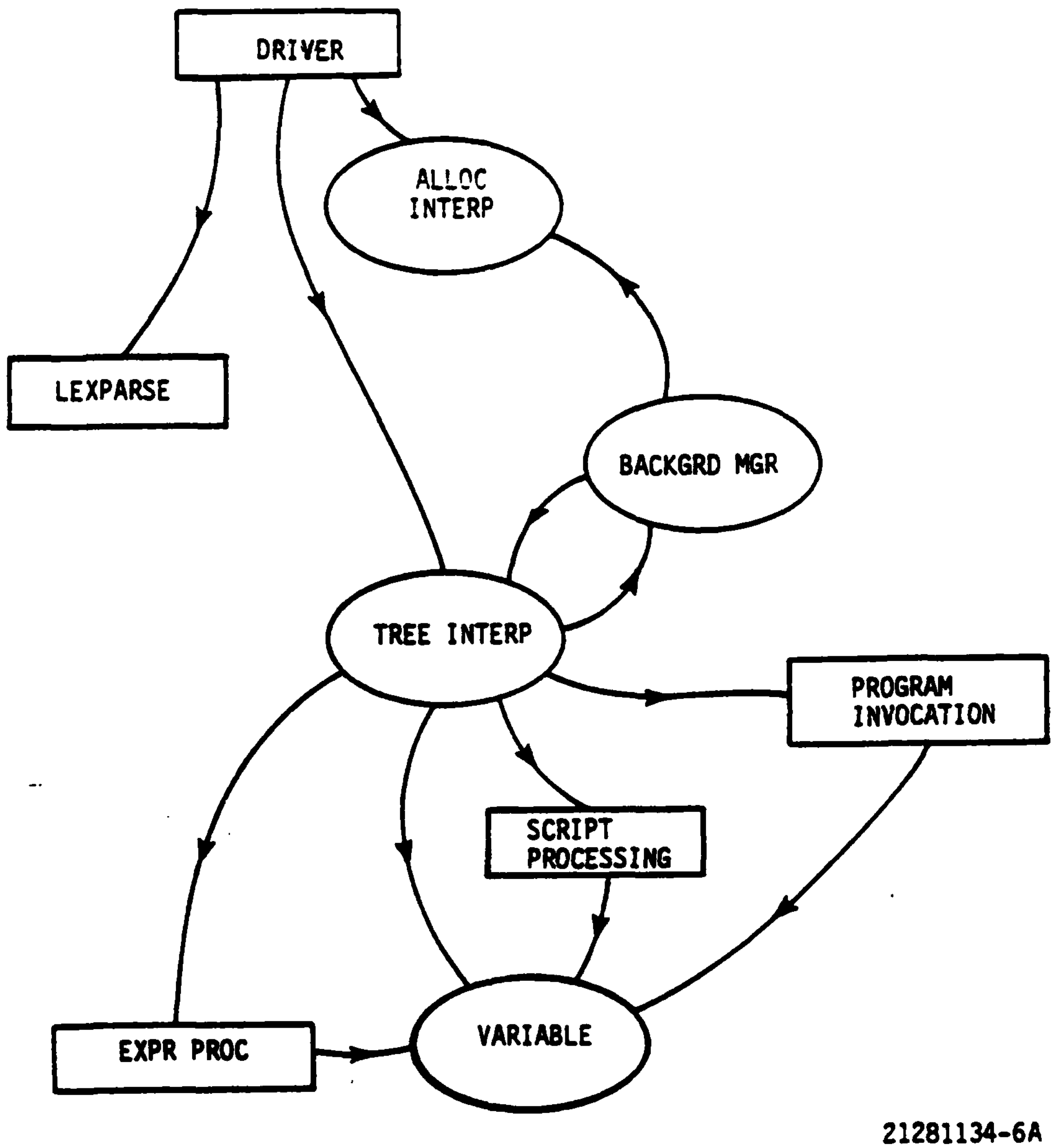


FIGURE 3-3: MCP MODULES

Arrows indicate flow of control.
Modules in rectangles are procedures or packages.
Modules in circles are tasks.
All Modules invoke ERROR.

SCRIPT PROCESSING manipulates script parameters. PROGRAM INVOCATION starts program execution. VARIABLE manages MCP variables. ERROR reports errors in MCL commands.

Any of these actions may manipulate MCP variables via the VARIABLE task.

3.3 Detailed Functional Requirements

A summary description of the MCL Language is presented below to give an overview of MCP functionality. This is followed by a discussion of the components of the MCP. A more detailed view of MCL is provided in Appendices A-C.

3.3.1 Command Input

The MCP interprets commands written in MCL and performs the actions they specify. The MCP command repertoire, summarized in Table 3.1, provides all standard functions required to run MAPSE tools and user programs. Commands can be executed in the foreground or background and can be executed as co-routines. The user can, via the MCP, interrupt and restart program execution, manipulate MCP variables and database objects, and direct the flow of control of program executions. Commands may be entered interactively or stored, as scripts, for later execution. Via the MCP, a user can invoke any Ada program, including itself. Further, any executing Ada program can invoke the MCP.

The MCL language borrows Ada syntax wherever possible. Appendix D summarizes those areas in which MCL syntax differs from that of Ada. In the following discussion, all tokens or constructs that are identical to Ada constructs are so noted with a parenthesized reference to the appropriate Ada LRM section. Their formal syntax (expressed via the variant of Backus-Naur Form used in the Ada LRM) appears in the complete syntax in Appendix C. Tokens or constructs that are MCP-specific are described in detail and their BNF representation included in the text. Interactions between a user and the MCP are included below for illustrative purposes. In such samples, characters printed by the MCP are underlined.

An MCL command consists of a sequence of tokens separated by blanks or carriage returns, and delimited by a semicolon (which can be omitted if it immediately precedes a carriage return). A command is optionally preceded by a label, of the form identifier:. Typing the interrupt character (control-C) causes all tokens processed in the current command to be ignored. If a command contains an error, the MCP issues an error message.

The MCP processes command input line by line, parsing a line when a carriage return is entered. Any commands delimited by semicolons are executed. If the tokens immediately preceding the carriage return describe a complete command, the MCP assumes that the semicolon was omitted, and executes the command. Otherwise, the MCP examines the next line for continuation text.

In interactive mode, a prompt is printed after each line is processed to indicate that the MCP is ready to process the next line. This may be a primary default prompt (':'), which indicates that the processed line ended with a complete command, or a secondary prompt ("line number/") to remind the user that command input is incomplete.

For example:

```
: COMPILE MYFILE        -- comments as in Ada
: %A := 2;              -- MCP variables begin with a '%'

: COMPILE MYFILE; LIST MYFILE


: LISTLOOP:  FOR %I IN 1..5
  2/ LOOP                    -- command not complete yet
  3/   LIST MYFILE
  4/ END LOOP
: -- primary prompt indicates the command is complete
```

Figure A-1 shows a sample MCP session.

When the user logs into the AIE, the MCP is invoked on his behalf. Associated with this MCP is a database context object that can be used to create temporary database objects, as well as a "current" window for permanent database objects, and a "home" window which serves as a user's home directory. See AIE(1).KAPSE(1) for a description of the database. The user may specify a sequence of commands to be performed by the MCP as part of its startup, by placing those commands in the database object "MCP_STARTUP" in TOP_LEVEL_DATA. This startup file enables the user to create a pre-defined environment in which to work. For example, "MCP_STARTUP" might contain the text:

```
%LIB:=ADALIB.YOURLIB.MYLIB

CHECK_MAIL                     -- program invocation

"DON'T FORGET TO CALL HOME" -- expression to be printed
```

The MCP, as part of its startup, would define the variable %LIB, invoke the CHECK_MAIL program, and print a message. It would then begin taking commands from the user.

TABLE 3.1: COMMAND SUMMARY

<table>
<tr><th>Command</th><th>Purpose</th><th>Example</th></tr>
<tr><td>help</td><td>provide help for a program or MCP script</td><td>HELP COMPILE</td></tr>
<tr><td>program call</td><td>invoke a program or MCP script</td><td>COMPILE MYFILE</td></tr>
<tr><td>assignment
get
put</td><td>expression manipulation</td><td>%A:= %B+2</td></tr>
<tr><td>loop
if</td><td>flow control</td><td>IF %A<2 THEN
PUT "OK" END IF</td></tr>
<tr><td>return
logout
suspend</td><td>terminate MCP processing</td><td>LOGOUT</td></tr>
<tr><td>resume</td><td>resume a previously suspended MCP session</td><td>RESUME MYMCP</td></tr>
<tr><td>-|</td><td>connect commands as co-routines via pipes</td><td>FLIGHT -| LANDER</td></tr>
<tr><td>->
-<</td><td>redirect a command's standard input or output</td><td>SORT MYFILE ->
OUTFILE</td></tr>
<tr><td>-&</td><td>execute a command in the background</td><td>COMPILE MYFILE -&</td></tr>
<tr><td>abort
wait</td><td>abort or wait for the completion of a background command</td><td>ABORT T4</td></tr>
<tr><td>block</td><td>group commands for I/O redirection, pipes, or background execution</td><td>BEGIN COMPILE A
LIST A
END -> OUTPUT</td></tr>
</table>

TABLE 3.1: COMMAND SUMMARY (Cont'd.)

Command	Purpose	Example
stop start cancel status	control the execution of an invoked program or MCP script	STATUS.T1.COMPILE
exec	interpret data as a command	EXEC %A & LOW
script specification	import and export parameters within a script	PROCEDURE COMPLIST (%FILE:STRING) IS BEGIN COMPILE %FILE LIST %FILE END

INTERMETRICS INCORPORATED • 733 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

3.3.2 DRIVER

3.3.2.1 Inputs

Driver accepts as input optional user parameters indicating a source of command input other than standard input. This can be either a string of MCP commands (COMMAND_STRING) or the name of a database object containing a command script (SCRIPT).

3.3.2.2 Processing

Since the MCP supports simultaneous command interpretation, (background tasks and pipe processing) it is necessary that some functions be performed by MCP-wide tasks. These functions include:

Global data management	(VARIABLE task)
Simultaneous command interpretation	(TREE INTERP task)
Task management	(ALLOC INTERP task)
Backgrd task name management	(BACKGRD MGR task)

These tasks are activated as part of the DRIVER initialization processing.

DRIVER determines the source of command input by examining the input parameters. Commands can be read from a script file or a string specified in the parameter list to MCP. If the parameters do not specify a script file or command string, command input will be read from standard input. If the MCP has been invoked to process a script, then DRIVER saves the script parameters internally, so they may be referenced later during the script header processing.

The DRIVER loops to process commands sequentially. It processes each command by:

(1) polling the BACKGRD MGR task to determine if any background tasks have terminated. If tasks have terminated, the user is notified;

(2) invoking LEXPARSE to parse the next command in the input stream into a parse tree; and

(3) invoking TREE INTERP to interpret the parse tree produced by LEXPARSE.

If a syntax or interpretation error occurs during command processing, tokens are flushed until the end of the script file or end of the terminal input line and an error diagnostic is issued.

Before termination, DRIVER places any OUT script parameters in the script's context object and processes a user shutdown command file, if any.

3.3.2.3 Outputs

The outputs from DRIVER are the script OUT parameter values. If the MCP was not processing a script, there are no output parameters.

3.3.3 LEXPARSE

LEXPARSE reads and parses MCL commands into a DIANA-like parse tree. The LEXPARSE package consists of two components: LEXICAL_ANALYZER and PARSE.

3.3.3.1 Inputs

LEXPARSE takes as its IN parameter an input stream from which commands are read.

3.3.3.2 Processing

PARSE is invoked to build a DIANA-like parse tree. This tree describes an MCL command, as well as any notation to be applied over the command. This latter category includes: REDIRECT_INPUT (-<), REDIRECT_OUTPUT (->), PIPE (-|) and BACKGRD (-&).

PARSE invokes LEXICAL_ANALYZER to read lexemes from the input stream. The parse algorithm is driven by tables generated from a LALR(1) grammar of the Mapse Command Language. The goal symbol of the grammar is a complete MCL command.

PARSE contains an exception handler for the interrupt exception, which causes the parse tree to that point to be ignored.

3.3.3.3 Outputs

LEXPARSE returns as its OUT parameter a DIANA-like parse tree. The tree is the internal representation of a single legal MCL command as parsed from the input stream.

3.3.4 ALLOC INTERP

The ALLOC INTERP task manages the family of TREE INTERP tasks activated by the DRIVER. A TREE INTERP task must be allocated every time a separate thread of execution in the interpretation of commands is needed. This is the case for the execution of co-routines connected through a pipe or for background execution of MCP commands.

3.3.4.1 Inputs

For allocation, no inputs are required. For deallocation, input is an access to the TREE INTERP task to be deallocated.

3.3.4.2 Processing

ALLOC INTERP has two entry points; one to allocate (ALLOC), and one to deallocate (DEALLOC) TREE INTERP tasks. ALLOC allocates a TREE INTERP task from a pool of active but quiescent TREE INTERP tasks. When the TREE INTERP task is completed, DEALLOC will be used to return the task to the pool for future use. The MCP will need a TREE INTERP task to process foreground commands, as well as a separate TREE INTERP task for each background command.

3.3.4.3 Outputs

ALLOC returns an access to a TREE INTERP task. DEALLOC has no outputs.

3.3.5 TREE INTERP

The parse tree interpreter is the heart of the MCP. It processes all MCL commands in the foreground, background, as co-routines and in scripts. Since several of these interpretations take place concurrently, TREE INTERP is a family of tasks of which members are allocated via ALLOC INTERP to interpret a single component of a co-routine or a background command.

3.3.5.1 Inputs

TREE INTERP takes as input: (1) a stream to be used as standard input for executing commands; (2) a stream to be used as standard output for executing commands; (3) the name of a composite object in which context objects for program invocation may be created, and (4) a DIANA-like parse tree (built by LEXPARSE) to be interpreted.

3.3.5.2 Processing

3.3.5.2.1 I/O Redirection

The TREE INTERP task first performs any I/O redirection that affects the command being interpreted. Each TREE INTERP task maintains a local copy of standard input and standard output. The streams passed as IN parameters serve as the default standard input and output if they aren't explicitly redirected. In the case of explicit redirection from/to database objects, the objects are opened and replace the local copy for standard input/output.

3.3.5.2.2 Pipes

The term "pipe" refers to the execution of concurrent commands such that the standard output of one command is directed ("piped") to the standard input of the next command. These two commands then form a "pipeline" and can be visualized as a single process with a standard input and standard output. The standard output from this pipeline could be used as the input to another command through another pipe, or redirected to a database object. A pipe is implemented by the KAPSE (see AIE(1).KAPSE(1), "Access Methods"). The MCP processes a pipe by creating the pipe file and allocating another TREE INTERP task to interpret the left side of the pipe. (The right side is processed by the current TREE INTERP task.) Each command within the pipeline does not need to know if it is accessing a pipe or database object; the opened pipe file or database object is passed to each TREE INTERP task as an IN parameter.

3.3.5.2.3 Program and Script Invocation

TREE INTERP generates a unique context object name and passes it to the PROGRAM INVOCATION package along with the parse tree. It is not necessary to distinguish between invoking a program and a script; the KAPSE will recognize the object is a script and invoke the appropriate interpreter/command processor. For parameterized scripts, the script header is passed to the SCRIPT package to import parameters. TREE INTERP recurses to interpret the script body. The SCRIPT package is called again when the body commands are executed or a RETURN command has been encountered. For background execution, the parse tree is passed to the BACKGRD MGR for interpretation.

3.3.5.2.4 Other Command Processing

(a) Assignment. The right hand side of an assignment command is evaluated by EXPR PROC and the resulting value is assigned to the variable or database object on the left hand side.

(b) RETURN, SUSPEND, LOGOFF. The POLL entry of BACKGRD MGR is called to determine if any background tasks are active, in which case the user is informed and the command is ignored. Otherwise the MCP terminates.

(c) ABORT. The B_ABORT entry of the BACKGRD MGR task is called to abort the specified background command task.

(d) WAIT. The WAIT entry of the BACKGRD MGR task is called to wait for the completion of the specified background command task.

(e) Expression. The expression tree is passed to the EXPR PROC package to be evaluated. The resulting value is written to the command's standard output.

(f) EXEC. The expression argument is evaluated by the EXPR PROC package and converted into a string. The string is used as the input stream to LEXPARSE to generate a new parse tree. TREE INTERP recurses to interpret the new tree.

(g) IF. Successive conditions are evaluated until one evaluates to TRUE. TREE INTERP recurses to execute the subtree associated with that condition. If none of the conditions evaluate to TRUE, no subtree is executed.

(h) CASE. The case expression is evaluated and compared to each value until the chosen alternative is found. TREE INTERP recurses to execute the subtree for that alternative. If none of the alternatives are acceptable, no subtree is executed.

(i) LOOP. The iteration clause is evaluated and TREE INTERP recurses to execute the loop body until the iteration condition has been fulfilled, or until a command within the loop terminates the loop (EXIT), or the MCP session (SUSPEND, LOGOFF, RETURN).

(j) EXIT. The immediately enclosing loop is terminated.

(k) USE. User specifies a new default composite object prefix. Subsequently, whenever a variable is begun with "%.", the default prefix will be substituted.

(l) RENAMES. The command symbol table is updated to reflect the new reserved word.

(m) Blocks. INTERPRET recurses to perform the commands within the block.

3.3.5.3 Outputs

Not applicable.

3.3.6 BACKGRD MGR

A background command is processed by activating a TREE INTERP task, associating a unique name to that task, and executing it. The task identifier will allow the user to synchronize with it, abort it, or locate its output.

3.3.6.1 Inputs

Input to BACKGRD MGR is a command tree (from PARSE) to be interpreted in the background.

3.3.6.2 Processing

First, a unique task identifier is created and associated by the START procedure with an allocated TREE INTERP task. The TREE INTERP task will be passed the task identifier as its composite object parameter. When TREE INTERP is ready to begin (once it initializes itself with the composite object), the command tree will be passed to it for interpretation.

The WAIT procedure synchronizes foreground with a given background command. WAIT will not return to the calling TREE INTERP task until the background task is finished interpreting its tree.

The B_ABORT procedure aborts a background command, terminating all active TREE INTERP tasks associated with the background command and calling upon ALLOC INTERP to deallocate them.

The POLL procedure checks the status of currently active background tasks. POLL determines the status of currently active tasks by issuing conditional entry calls. Active task and "just terminated" background command status is returned.

3.3.6.3 Outputs

Not applicable.

3.3.7 EXPR PROC

EXPR PROC evaluates expressions written in MCL.

3.3.7.1 Inputs

Input to EXPR PROC is an MCL expression in parse tree form.

3.3.7.2 Processing

EXPR PROC evaluates the tree by returning actual values of primitive (leaf) nodes (e.g., literals, variables), calling itself recursively on children of operator or function nodes, and then calculating the result of the operation or function with the returned values of the children.

Expression operators fall into the following classes: arithmetic, boolean, string, and comparison. Also evaluated are attributes LENGTH TYPE, and type compatibility attributes. Finally, EXPR PROC will evaluate component selection of composite objects, i.e., arrays and records.

3.3.7.3 Outputs

An MCL value is returned to the calling package or task.

3.3.8 SCRIPT

3.3.8.1 Inputs

Inputs to SCRIPT consist of script header IN parameters.

3.3.8.2 Processing

The SCRIPT package is called by TREE INTERP to process the script header. SCRIPT imports any input parameters and returns output parameters so that the PROGRAM INVOCATION package can pass/receive values to/from scripts. (A script may be invoked with input parameters. These must be evaluated and substituted for script formal parameters prior to execution of the script.)

A SET PARAMETERS routine imports parameter values as necessary by finding the IN parameters values stored internally by DRIVER at Startup time; associating them with parameters named in the Script header, either by name or by position; creating an MCP variable with the same name for each parameter in the script header; initializing the value of any input parameter's variable created above to the corresponding input value (with SET in the VARIABLE task); initializing any input parameter variable which does not have a corresponding input value by its default, located in the script header. On completion of the script, an OUT PARAMETERS routine will reverse this, taking values from the MCP variables that correspond to OUT parameters in the Script header and setting each output parameter to its corresponding variables value. Any parameter which is passed for input but not specified in the script header will cause a default out parameter to be created and initialized to the input value. Scripts which are functions return their value as a parameter named RETURN.

3.3.8.3 Outputs

Not applicable.

3.3.9 PROGRAM INVOCATION

PROGRAM INVOCATION calls on executable programs and scripts located in the database. It is activated by a Program Invocation command. Its input parameters are: a parse tree of the Program Invocation command and the name of a context object to be created for the invocation.

3.3.9.1 Inputs

PROGRAM INVOCATION accepts as input a parse tree of the program invocation command and the name of a context object to be created for the program invocation.

3.3.9.2 Processing

PROGRAM INVOCATION must evaluate its input parameters; invoke the appropriate program via a KAPSE call, passing the parameter values; wait for the program to complete (unless program is invoked directly in the background); update all OUT parameters to reflect their values on termination of the program (export their values from the invoked program to the parent MCP).

IN parameter values are contained in the parse tree. The program or script name to be executed is expanded into a full database object name by a KAPSE routine. If the object is a program, KAPSE invokes it. If it is a script, KAPSE invokes the proper interpreter (the MCP itself, if it is an MCL script) to process that script, and the script file name itself is the first input parameter. A KAPSE call will be made to await program completion, after which all OUT parameters values will be obtained and the variables corresponding to those parameters will be updated by SET (in package VARIABLE). A program completion status is output.

OUT parameters are obtained as follows. A KAPSE call is invoked to return a parameter string. For each parameter in the string, the parse tree of the program invocation command is examined for a parameter which corresponds either positionally or by name. If one is found, it is updated to contain the value of the variable in the returned parameter string. If none is found, a default OUT parameter will be generated, and its value set as above.

3.3.9.3 Outputs

PROGRAM INVOCATION outputs the status of the invoked program.

3.3.10 VARIABLE

VARIABLE maintains command processor variables in an internal variable space. It is responsible for allocation, alteration, and assignment to/from variables via the VARIABLE primitives SET and FETCH.

3.3.10.1 Input

VARIABLE accepts as input: (1) an MCP variable name for FETCH, or (2) an MCP variable name and the value to be associated with it for SET.

FETCH finds a value for a variable name and looks up the given name in a symbol table. If the variable is found, it's value is returned, otherwise the NULL value is returned to the calling package or task.

SET associates a MCP variable name and a value. IF the variable did not previously exist, it is created implicitly by the MCP in the internal variable space.

3.3.10.2 Outputs

VARIABLES outputs the value of the specified variable for FETCH. For SET, there is no output.

3.3.11 ERROR

ERROR reports error messages and severities. It may be invoked from any aforementioned package or task.

3.3.11.1 Inputs

ERROR outputs an error code, a message string (optional) and a severity code.

3.3.11.2 Processing

For each error code there is an associated error message; that message will be printed to standard output. The severity of the error, passed in the severity code parameters, will also be posted. If a message string is also passed as an input, it will also be printed. Typically, it will report the location of the error in the erroneous command, or the lexname which is erroneous.

Finally, a COMMAND ERROR exception will be raised in the inner most TREE INTERP task enclosing the task or package which invoked ERROR.

3.3.11.3 Outputs

None.

3.4 Adaptation

The following parameters may be supplied as part of MCP installation:

1. The maximum number of concurrently executing background tasks.
2. The maximum amount of storage to be allocated to the variable symbol table.
3. The maximum amount of storage to be allocated to a single command parse tree.

3.5 Capacity

A command line exceeding 255 characters will cause a SEVERE error message to be generated. A compound MCL statement is limited to roughly 2000 lines of substatements (this limit is dependent on the maximum amount of space that may be allocated to a single command parse tree). All variable data space may not exceed a limit to be determined.

4.0 QUALITY ASSURANCE PROVISIONS

4.1 Introduction

Testing of the Mapse Command Processor will be performed at three levels.

1) Unit testing of the component parts

2) Integration testing of the component parts within the MCP

3) Functional testing of the MCP subsystem. The functional test approach described below will be repeated on both AIE configurations (IBM 4341 and PE 8/32).

4.2 Test Requirements

This section describes the requirements for each of the three levels of testing.

4.2.1 Unit Testing

Each module internal to the MCP, as described in the functional specification, and each subunit thereof will be tested prior to integration. Testing will be implemented with the following strategy. Small driver routines will be written for each unit to be tested. The driver routines will be run many times with a series of inputs to the invoked units. These inputs will either span the entire set of possible inputs to the given unit or, if such a set is prohibitively large, a representative selection of typical usage and "worst-case" inputs will be tested. When primitive subunits have been thoroughly tested and the interfaces have been integration tested, the combined unit will be unit tested.

Modules will be unit tested in the following order, which reflects dependencies: ERROR, LEXPARSE, VARIABLE, EXPR PROC, ALLOC INTERP, TREE INTERP, PROGRAM INVOCATION, BACKGRD MGR, DRIVER.

4.2.2 Integration Testing

Integration Testing will be performed to ensure reliability of unit interfaces. Once the units have been tested, integration tests will be performed by using the invoking unit to call the invoked unit, and examining input and output parameters. Alternatively, stubs may be used to replace the invoked units. Their purpose will be to report the values of all input parameters. In this way it may be verified that the subunits are being invoked from calling units with the proper parameters.

INTERMETRICS INCORPORATED • 733 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

4.2.3 Functional Testing

Functional tests will be defined to verify the requirements contained in this specification. Formal functional testing will be performed in three parts, testing: (1) the interaction of the MCP with other tools of the MAPSE, (2) the command language lexical and syntactic elements, and (3) the command statement processing.

The first part tests the MCP-KAPSE interface. The MCP interfaces with other tools in the MAPSE through KAPSE primitives and the database. Tests will check:

Program invocation of other tools.
Parameter processing and function return values.
Database object manipulation

The second part of the testing verifies that the language elements (lexemes and syntax) correspond to the specification. Tests will be performed to verify legal lexemes. Since the lexemes of MCL closely resemble those of Ada, tests such as the Ada Compiler Validation Facility can be slightly modified and used to test the MCP. Tests will be performed as language scripts so that they may be generated and run automatically. The interactive use of the language will be tested by typing a few of the tests at the terminal and checking that the same results are generated.

The third part of the formal tests verifies the processing of statements done by the MCP. Tests will check:

I/O redirection
Background execution
Expression evaluation and manipulation
Flow control and block statement
Startup and Termination processing

4.3 Acceptance Test Requirements

Acceptance tests will be conducted to ensure that the MCP conforms to the general requirements. These will include formal tests (section 4.2.3) and capacity tests designed with the maximum number of concurrently executing background tasks, using maximum size symbol table, using maximum number of characters allowed on a command line and using very long command statements. In addition, system testing will be executed using interactive MCL and scripts, and will include acceptance testing for MCP.

APPENDIX A THE MCL COMMAND LANGUAGE

The following paragraphs describe the commands available to the AIE user via the MCP. Figure A-1 shows a sample MCP session.

A.1 Help Command

Information about invokable programs may be obtained via the HELP command.

HELP with no parameter requests general information about all MCP commands and invocable programs. Information about a specific program can be requested by providing the program name as an argument to HELP. If the program specified does not exist or help is not available for it, the user is informed. Notice that a program name is specified as an unquoted string. Within commands, the MCP interprets program names in terms of the PROGRAM_SEARCH_LIST attribute associated with the MCP's context object. The program must lie within one of the composite objects named in PROGRAM_SEARCH_LIST.

A.2 Program Invocation

A program invocation is made by specifying the name of the program to be executed after a prompt, followed, as required, by a list of parameters.

The syntax for a program call is similar to Ada syntax for a procedure call, except that: (1) the procedure name is replaced by the name of the program to be invoked; and (2) the comma between parameter associations may optionally be replaced by a blank. However, if the user wishes to continue the parameter list on the next line, a comma is required after the last parameter association on the current line.

A.2.1 Parameter Passing

The actual parameter part of a program call consists of one or more parameter associations, each of which specifies an actual parameter to be passed to the invoked program, either positionally or by name.

```
                    WELCOME TO THE MAPSE SYSTEM

: HELP COMPILE             -- command help
        .
        .
        .
     help text
        .
        .
        .
: COMPILE MYFILE LIB => MYLIB -- compile program invocation with
                           -- positional and named parameters

: FOR %I                   -- MCP variable names start with '%'

  2/ IN 1..10              -- secondary prompt

  3/ LOOP                  -- flow control.
                           -- LPT is a program which queues text
  4/ LPT MYFILE            -- files to be printed on the line printer

  5/ END LOOP

: FLIGHTSIM LEVEL=>3       -- another program invocation

: %STATUS.EXIT_STATUS      -- display exit status of the last invoked
                           -- foreground program
     OK

: LOGOFF
```

FIGURE A-1: SAMPLE MCP SESSION

For positional parameters, the actual parameter corresponds to the formal parameter with the same position in the invoked program's formal parameter list. For named parameters, the corresponding formal parameter is explicitly given in the call. As opposed to Ada, positional and named parameters may be freely mixed. The MCP parses the parameter association by grouping together all positional parameters, followed by all named parameters. For named parameters, the MCP convention is that only the last occurrence of a parameter association is used, permitting a user to change a mistyped named parameter without retyping others. For example, the line:

```
: FLIGHT  3 5 LEV=>2 6 7 LEV=>3
```

associates the value 3 with LEV.

An invoked program may associate a mode of IN, OUT, or IN OUT with a formal parameter. An actual parameter of mode IN OUT or OUT should be represented by a variable name. If it is not, a warning is printed, and the parameter is ignored. (If its mode is IN OUT, its initial value will be used). The specified variable's value is updated when the program completes its execution.

As in Ada, if a program's declaration specifies a default value for an IN parameter, then the corresponding parameter may be omitted from a call.

MCL also implements default OUT parameters if the user fails to specify an actual parameter of mode OUT, or if the specified parameter was not a variable name. In these cases, the MCP generates an implicit variable declaration. The generated variable's name is the catenation of '%' and the formal parameter name. For example, if a program COMPILE's main subprogram had the specification:

```
procedure compile (file: string; lib: string;
                   max_error_severity:  out string);
```

the MCP user could type

```
: COMPILE MYFILE MYLIB
```

At the conclusion of COMPILE's execution, an MCP variable named %MAX_ERROR_SEVERITY would be generated and assigned the OUT parameter value. The MCP user will be informed of any variables generated via default OUT parameters if the pre-defined MCP environment component INFORM_DEFAULT_OUT is TRUE (Section A.10).

A default OUT parameter may generate a variable name which conflicts with a currently existing MCP variable. For example,

```
: %MAX_ERROR_SEVERITY:= OK
: COMPILE MYFILE MYLIB
: --default OUT parameter variable
: --%MAX_ERROR_SEVERITY is already defined
```

If the pre-defined MCP variable component %ENVIRONMENT.AUTO_REDEFINE is TRUE, the OUT parameter value replaces the current value of the variable. If the %ENVIRONMENT.AUTO_REDEFINE component is FALSE, the out parameter value is ignored.

The user may request information about a positional or named parameter by supplying a question mark in place of the actual parameter value, followed by a new line. The action taken depends on the manner in which the invoked program chooses to deal with requests for parameter help. If the program can provide no parameter help, the user is informed, and may continue specifying parameter associations to the program.

If the called program provides help text for the relevant parameter, then that text is printed, and the user may continue specifying parameters. If the called program is set up to interpret any help requests itself, the actual parameter part is considered complete, and the program is invoked. At the conclusion of the program's invocation, the user will be allowed to resubmit his program, this time presumably with actual parameters.

A.2.2 Function Invocation

An MCL function is a program that returns a value. Like program, a function is invoked by name and supplied any necessary parameters. A function call's actual parameter part is similar to that of a program call, the differences being that it is surrounded by parentheses, it can only contain IN parameters, and no whitespace is permitted between the function name and the left parentheses. The user may also request parameter help for a function.

Examples of function calls:

```
STRING(%V)

SINE(2.0)
```

A.3 Expression Manipulation Commands

MCL includes commands that: store an expression's value in a variable or database attribute (assignment); read and store a user-typed literal (GET), and display the value of an expression (PUT).

A.3.1 Assignment

The assignment command (:=) replaces the current value of an MCP variable or database object attribute with the value of an expression.

Since all MCP options are controlled via pre-defined components of the %ENVIRONMENT variable (Section A.10), the assignment statement can be used to modify these options.

Examples:

```
: %V1:="HELLO"

: %V1:=HELLO                          -- equivalent to above,
                                      -- since identifiers don't
:                                     -- have to be surrounded
                                      -- by quotes.

: .MYLIB.MYFILE'CONFIG := 4           -- an attribute (named
                                      -- CONFIG) of the
                                      -- database object
                                      -- MYLIB.MYFILE
: %ENVIRONMENT.PROMPT := "%"          -- modify MCP prompt

%
```

A.3.2 GET

The GET command causes the MCP to read an arbitrary sequence of literals from standard input and store them in MCP variables. The literals must describe values of valid MCL types, and must be separated by blanks, commas, or newlines. Examples:

```
: GET %V1 %V2       -- read values from standard input

     "V1's VALUE"

     2.0
```

A.3.3 PUT

The PUT command causes the MCP to print the values of an arbitrary sequence of expressions to standard output. For example:

```
: PUT %A %B

: PUT "THIS STRING IS PRINTED" & " TO STANDARD OUTPUT"

: PUT HELLO     -- unquoted string
```

The keyword "PUT" may be omitted when a single expression is to be displayed except in the case of an unquoted string, which is assumed to be a program call. For example:

```
%STATUS.EXECUTION_TIME

  5.2

: %X = %Y

  FALSE
```

A.4 Database Commands

A variety of KAPSE primitives such as "delete" and "copy", may be invoked directly through the MCP. See AIE(1).KAPSE(1) for a complete list of database manipulation facilities.

A.5 Control Commands

Commands in an MCP script are normally executed in the sequence in which they appear. The sequence of command execution can be altered via MCL control commands.

A.5.1 IF

The IF command selects for execution one or none of a sequence of MCL commands, depending on the value of one or more corresponding conditions. The syntax for the IF command is identical to that of Ada. The expressions specifying conditions must be of the predefined type BOOLEAN.

A.5.2 Loop

A loop command specifies that a sequence of statements is to be executed zero or more times. As in Ada, an iteration clause (see Appendix B) optionally precedes the loop. The loop command, like all MCP commands, may be executed in interactive mode as well as the batch like modes background and script.

Note that an EXIT command causes the termination of an enclosing loop.

An exit command may only appear within a loop.

A loop command without an iteration clause specifies repeated execution of the basic loop. The basic loop may be left via an interrupt, via an EXIT command, or if a command within it terminates the MCP session.

If a loop command contains a WHILE clause, the condition is evaluated and tested before each execution of the basic loop. If it evaluates to FALSE, the loop statement terminates; otherwise, the basic loop is executed.

If a loop command contains a FOR clause, the range is evaluated once before the execution of the basic loop. If the range is an aggregate value, the loop parameter sequences through the components as the loop is executed. Examples:

```
:  FOR %I IN 1..4  LOOP

   2/ GET %A

   3/ IF %A /= "" THEN FLIGHTSIM %A END IF

   4/ END LOOP
```

A.6 MCP Termination

There are several ways to terminate MCP processing. In all cases, the MCP must be quiescent (i.e., no background commands can be active); otherwise, the user is warned and the terminating command is ignored.

A.6.1 RETURN

The RETURN command terminates the execution of the MCP, and returns control to the MCP's invoker. If the MCP is processing a script that describes a function, the RETURN command must include an expression whose value is the result returned by the function.

A.6.2 LOGOUT

The LOGOUT command also terminates the execution of the MCP. Unlike RETURN, LOGOUT causes the termination of interactive session. Control is returned to the system initiator.

A.6.3 SUSPEND

The SUSPEND command terminates MCP processing, maintaining its current context, and returns control to the invoker. It requires as an argment, the name of a database object in which the MCP's context is to be saved for reference by a subsequent RESUME command.

```
:  SUSPEND MYMCP

 MCP SESSION SUSPENDED

 MAPSE SESSION TERMINATED -- typed by the MCP's invoker which
                          -- was the System Initiator
```

A.6.4 Shutdown Commands

The user may specify a sequence of MCL commands to be performed automatically prior to MCP termination via LOGOUT or RETURN. These commands must be contained in the database object "MCP_SHUTDOWN" in the MCP's TOP_LEVEL_DATA window.

For example, "MCP_SHUTDOWN" might contain the text:

```
PUT %STATUS
LIST MYFILE
```

A.6.5 RESUME

The RESUME command resumes a previously SUSPENDed MCP session. The name under which the context of a previously SUSPENDed MCP was stored must be supplied as an argument. Example:

```
: RESUME MYMCP

: --this prompt is printed by the resumed MCP session.
```

A.7 I/O Redirection

Many MCP commands (for example, PUT) read from standard input or write to standard output. If unmodified, a command's standard I/O defaults to the MCP's own standard input and output. The user may optionally redirect the standard input or output of any command.

A.7.1 Standard I/O

The notation -< is used to redirect a command's standard input. The notation -> is used to redirect a command's standard output. The specified database object is opened (for standard input) or created (for standard output). The command is interpreted, and the database object is closed. For example:

```
: FLIGHTSIM -< CONTROL_FILE ->RESULTS
                  --FLIGHTSIM reads its standard input from
                  --CONTROL_FILE and writes its standard output
                  --to RESULTS

: COMPILE MYFILE LIB=>MYLIB ->RESULTS
```

Notice that the second output redirection to RESULTS caused the first command's output to be overwritten.

A.7.2 Pipes

The notation -| between two commands indicates that the commands are to be connected via a pipe. Standard output of the command to the left of the -| becomes the standard input of the command to the right of the -| and the commands execute as co-routines. A sequence of commands connected via pipes is referred to as a pipeline command.
For example:

```
: PUT "STRING TO SERVE AS STANDARD INPUT" -| LPT
                --string is queued to
                --be printed by line printer

: DATE -| FLIGHT_LOG OPTION=>HEADER
```

A.8 Backgrd Execution of Commands

An MCL command is normally executed in the foreground; i.e., the MCP waits for the command to complete its execution before accepting further command input. To specify that a command is to execute in the background, the user terminates a command with the notation -&. In this mode, the MCP begins the execution of the command, but does not wait for it to terminate before accepting subsequent user commands. For example:

```
: COMPILE MYFILE LIB=>MYLIB -&
```

A background command can be viewed as a task object that executes asynchronously. This object has a name, of the form "TX", where X is an integer incremented for each task generated. Alternatively, if the background command is preceded by a label, the task name is equivalent to the label name. In either case, the name is displayed to the user when the task begins its execution, and is also stored in the pre-defined MCP variable component %STATUS.LAST_TASK. For example:

```
: COMPILE MYFILE LIB=>MYLIB -&

  T4 executing

: -- ready to accept next command

: MYTASK: FOR %I in 1..4 LOOP

  2/ FLIGHTSIM %I END LOOP -&

  MYTASK executing

:
```

The user is informed when the task completes its execution. For example:

MYTASK completed

:

The task name can be used to refer to a task in order to abort it, via the ABORT command.

The user may begin the execution of a command in the background, then decide to wait for it to conclude its execution. The WAIT command causes the MCP to wait for the conclusion of the specified task(s) before accepting further command input from the user. For example:

```
: COMPILE MYFILE LIB=>MYLIB -&

  T4 executing

: ABORT %STATUS.LAST_TASK

  T4 aborted

: COMP: COMPILE YOURFILE -&

  COMP executing

: WAIT COMP

  COMP completed

:
```

A pipeline command may also be invoked in the background. For example:

```
: FLIGHTSIM -| SORT -&

  T6 executing

:
```

A.9 Compound Commands

Multiple commands may be grouped together in a block in order to apply a notation to all of them via the BEGIN and END block commands. For example, the block

```
:  BEGIN
  2/ LIST MYFILE
  3/ COMPILE MYFILE LIB=>MYLIB
  4/ END -| LPT
```

pipes the catenated output of LIST and COMPILE to the LPT program.

The block commands may be applied to -< and ->. The specified database object is opened (for standard input) or created (for standard output) and remains open for the duration of the block.

For example, the block

```
:  BEGIN
  2/ LIST MYFILE
  3/ COMPILE MYFILE LIG=>MYLIB
  4/ END -> .TMP
```

redirects the catenated output of LIST and COMPILE to the .TMP database object.

The block command also allows the user to group together commands for sequential background execution. For example:

```
:  BEGIN
  2/ COMPILE MYFILE LIB=>MYLIB
  3/ FLIGHTSIM
  4/ END -&
  T12 executing
:
```

causes COMPILE to be executed in the background, followed by FLIGHTSIM. Notice that this is different from:

```
: COMPILE MYFILE LIB=>MYLIB -&

  T12 executing

: FLIGHTSIM -&

  T13 executing
```

in which COMPILE and FLIGHTSIM execute simultaneously.

A.10 MCP Variables

The MCL user does not explicitly declare variables. Rather, a variable is implicitly declared by its first use. Its value is a string, with implicit conversions performed as required. The pre-defined command DUMP_VARS prints to standard output a list of all MCP variables and their current values, separated by blanks. This is useful for determining the current state of all MCP variables.

A variable may contain an aggregate (Array or Record) value. The user may choose a particular component of the aggregate by means of the component select operations. A select operation may be applied to a variable on the left or right side of an assignment. Components which are not present are evaluated to the null string "" when selected on the right side of an assignment. If a named component on the left side of an assignment is not already a component of the variable, it is added. Position components may be appended to the aggregate variable by specifying one greater than the last position. For example:

```
: %X := (A=>1, B=>2)

: %X.A := 3 -- Results in (A=>3, B=>2)

: %X.C := 4 -- Results in (A=>3, B=>2, C=>4)

: %Y := (100, 200)

: %Y(3) := 300 -- Results in (100, 200, 300)
```

A.10.1 MCP Variable Attributes

The following attributes are defined for MCP variables. TYPE, INTEGER, REAL, STRING, BOOLEAN, ARRAY, RECORD, LENGTH. These attributes are preceded by a tic, e.g., %VAR'LENGTH. TYPE returns the variable's type: integer, boolean, string, real, array, or record. Attributes INTEGER, REAL, BOOLEAN, and STRING will return TRUE if the given variable is convertible (see closely related types, Appendix B.3.3) to the given attribute type. Otherwise, FALSE will be returned. ARRAY returns TRUE if its argument is an array; RECORD returns TRUE if its argument is a record. It should

be noted that, unlike Ada, all arrays are records in MCP. (See Aggregate definition in Appendix B.2). LENGTH returns the length in characters if its argument is a string, and the number of components if a record or array.

A.11 MCP Predefined Variables

The MCP predefines two RECORD variables named %ENVIRONMENT and %STATUS whose components have special meaning to the MCP. The components of %ENVIRONMENT control MCP options. The components of %STATUS serve as placeholders for values generated during command processing. When the MCP starts up, each %ENVIRONMENT component has an initial value. If the user redefines an option or %ENVIRONMENT illegally, the initial option value is used. Tables A.3 and A.4 list the component names, types, effect and initial value for each of the predefined variables.

A.12 External Program Invocation

Some MCL commands are executed via internal MCP actions (e.g., assignment) while others cause the MCP to invoke external programs. This latter category includes a program or function call and a script invocation. Each invoked program has associated with it a context object, which can be referenced to control the program's execution and determine its status.

A.12.1 Context Object Names

Each background task creates, within the MCP's context object, a composite object whose name corresponds to the task name. Within this composite object, the task creates a context object for each program it invokes. The name of each context object is equivalent to the name of the invoked program. If the same program is subsequently invoked within the same task, its context object name is of the form "program_nameX", where X is an integer starting from 2 and incremented for each repeated invocation of the program within the task. For example, the commands:

```
: COMPILE MYFILE LIB=>MYLIB -&

  T4 executing

: COMP:  BEGIN COMPILE MYFILE

  2/ COMPILE YOURFILE END -&

  COMP executing

:
```

TABLE A.3: MCP PREDEFINED VARIABLE (%ENVIRONMENT COMPONENTS)

NAME	TYPE	PURPOSE	INITIAL VALUE
PROMPT	STRING	Defines the MCP user prompt.	":"
INFORM_DEFAULT_OUT	BOOLEAN	If TRUE, the user is informed of each default OUT parameter variable generated.	FALSE
AUTO_REDEFINE	BOOLEAN	IF TRUE, a default OUT parameter will replace an already existing variable with the same name.	FALSE

TABLE A.4: MCP PREDEFINED VARIABLES (%STATUS COMPONENTS)

FCONTEXTS	STRING	The names of all context objects within the last foreground command.	""
EXECUTION_TIME	REAL	The execution time of the last completed foreground job command.	0.
EXIT_STATUS	STRING	The exit status of the last completed foreground job command.	""
LAST_TASK	STRING	The name of the last background task.	""
ACTIVE_TASKS	STRING	A list of all currently executing tasks.	""

would cause the creation of composite objects named ".T4" and ".COMP", and context objects named ".T4.COMPILE", ".COMP.COMPILE" and ".COMP.COMPILE2".

Foreground MCP execution can be thought of as a task object named "T1" which interacts with the user and executes commands synchronously. Context objects for any foreground programs are thus created in the composite object ".T1". The predefined MCP variable %FCONTEXTS contains a concatenated list of context object names created in ".T1" for the last foreground job. For example:

```
: COMPILE MYFILE LIB=>MYLIB

: %STATUS.FCONTEXTS

   .T1.COMPILE
```

The pre-defined MCP variable %STATUS.ACTIVE_TASKS contains a concatenated list of task names for all active tasks.

(b) Program Context Object Attributes: A program's context object has various attributes that give information about the program's execution (Table A.5). The context object may be treated as a database object for the purpose of referencing these attributes. A complete description of context objects is given in AIE(1).KAPSE(1).

The predefined variables %STATUS.EXECUTION_TIME and %STATUS.EXIT_STATUS contain the execution time and exit status for the last foreground command. If a program completes its execution with an exit status of "OK", all database objects within its context object are deleted.

Examples:

```
: COMPILE MYFILE LIB=>MYLIB

: %STATUS.EXIT_STATUS

    OK

: COMPILE YOURFILE LIB=>MYLIB -&

  T7 executing

: PUT .T7.COMPILE'TERMINATED

  FALSE
```

TABLE A.5: CONTEXT OBJECT ATTRIBUTES

Attribute Name	Possible Values	Meaning
TERMINATED	"TRUE" "FALSE"	Indicates whether the program's execution has completed.
EXECUTION_TIME	a string describing a REAL	Total execution time for the program (defined only if TERMINATED = TRUE)
EXIT_STATUS	"OK", "CANCELLED" "INTERRUPTED", or the name of an unhandled exception	The program's exit status (defined only if TERMINATED = "TRUE")

A.12.3 Control Commands

The MCL context object commands described below, take as arguments a sequence of context object names or the names of composite objects containing context objects, and perform actions on the programs they represent.

STOP. The STOP command is invoked to stop the execution of an arbitrary executing program. STOP takes as its argument one or more context object names.

START. The START command allows the user to restart the execution of a command suspended by a prior STOP command. The specified program(s) resume their execution in the background.

CANCEL. The CANCEL command is invoked to terminate the execution of one or more programs.

STATUS. The STATUS command enables the user to query the status of programs.

If a program within the list is currently active, the user is given the following information about its execution: execution time, memory used, I/O count. If the program is terminated, its total execution time and exit status are displayed.

Examples:

```
: F:  FLIGHTSIM -| LANDER -&
  F executing
: STOP .F.FLIGHTSIM
: DEBUG  .F.FLIGHTSIM
          .
    debugging
          .

: START .F.FLIGHTSIM
: STATUS .F.FLIGHTSIM
          .
          .
          .
    status information
          .
          .
          .
: CANCEL .F -- all context objects
            -- within .F, i.e,
            -- .F.FLIGHTSIM and
            -- .F.LANDER
```

A.13 EXEC

The MCP normally reads user-typed strings and interprets them as commands. The MCP is also capable of interpreting data as a command, via EXEC. EXEC takes as its argument an expression that must yield a string value that is recognizable as one or more MCL commands. For example:

```
: %A := "COMPILE MYFILE LIB=>"

: EXEC %A & MYLIB -- the command COMPILE MYFILE LIB=>MYLIB is

:                 -- executed

:  EXEC %A & YOURLIB -- COMPILE MYFILE LIB=>YOURLIB
```

EXEC's can be nested. For example

```
: %A:= "EXEC ""BEGIN COMPILE MYFILE;"" & %B &""MYFILE END->.TMP"""

: %B:= LIST

: EXEC %A   -- BEGIN COMPILE MYFILE; LIST MYFILE END->.TMP

: %B:= DELETE

: EXEC %A   -- BEGIN COMPILE MYFILE; DELETE MYFILE END->.TMP
```

A.14 RENAMES

Several reserved keywords have special significance in the MCL. Whenever appropriate, Ada reserved words have been used. In instances where new keywords are needed, the command language predefines identifiers to serve as keywords. These predefined identifiers may be redefined using the RENAMES command. For example:

```
: EX RENAMES EXEC

: EX %C
```

A.15 Nested MCP's

When the user logs in to the AIE, the MCP is automatically invoked on his behalf. Since this MCP is capable of invoking any program, it may also invoke itself.

```
: MCP

: -- This prompt is typed by the nested MCP.
```

The user may then issue commands to the invoked MCP, which is referred to as a nested MCP. The nested MCP has its own context object,and any changes made to variables in the nested MCP are not reflected in the invoking MCP.

If a nested MCP session is terminated by the MCP commands RETURN or SUSPEND, control is returned to the invoking MCP. If the LOGOFF command is issued, the MAPSE session is terminated (including the invoking MCP).

A nested MCP inherits the standard input and output of the invoking MCP. It reads commands from its standard input, and commands may in turn read from standard input or write to standard output as part of their execution. As with any invoked program, a nested MCP's I/O may be redirected via -> or -<. If the MCP's standard input is not the teletype an end-of-file is equivalent to RETURN. For example, if a database object MCP_INFILE contained the text.

```
FOR %I IN 1..5 LOOP GET%A; COMPILE%A; LIST%A END LOOP;

MYFILE YOURFILE F1 F2 TESTFILE -- This data in
                               -- standard input will be read
                               -- by GET
```

the user might type

```
: MCP -< MCP_INFILE
```

The MCP can optionally be supplied with a string which is to serve as its standard input, via the parameter INPUT_STRING. For example:

```
: MCP COMMAND_STRING => "COMPILE MYFILE; %STATUS.EXIT_STATUS"
```

Since the MCP is an ordinary tool, it can be invoked from any other program. The COMMAND_STRING parameter provides a convenient means of invoking the MCP within a program, specifying to it the command(s) it is to execute and data for those commands.

A.16 Scripts

A script is a sequence of MCP commands stored in a database object. It is functionally equivalent to a linked executable program in terms of invocation syntax, help information and parameter passing. Any command within the script may read from standard input or be written to standard output. For example, if a script named "SIM" contained the text.

```
"HOW MANY TIMES SHOULD THE SIMULATION BE RUN?"

GET %TIMES

For %I IN 1..%TIMES LOOP
```

```
        FLIGHTSIM OPTION=>FAST LEVEL=>3

    END LOOP
```

the user might type

```
    : HELP SIM
          .
          .
          .
       help text
          .
          .
          .

    : SIM
```

HOW MANY TIMES SHOULD THE SIMULATION BE RUN?

```
    10
    : -- The script has completed its execution.
```

A script differs from an input file for a nested MCP in that it contains only commands. The data for these commands comes from standard input, not the script. This is in keeping with the model of scripts as equivalent to linked executable programs.

A script can have its I/O redirected or be executed in the background. For example, if a database object "SIMIN" contained the text "10", the user might type:

```
    : SIM -< SIMIN -&
```

A script's exit status and execution time are available to the user in a manner identical to that for programs. For example:

```
    : SIM -< SIMIN

    : %STATUS.EXIT_STATUS

      OK
```

A script is terminated when end-of-file is reached, or when a RETURN, LOGOUT or SUSPEND command is encountered.

A.16.1 Script Parameters

A script may receive IN parameter values and return updated OUT parameter values if it contains a script specification command.

The script specification command simulates the execution of a parameterized Ada subprogram. It consists of a subprogram specification that gives parameter information, and a script body, which contains MCP commands to be interpreted in order to simulate the subprogram's execution. If a script specification command is given, it must be the first line of the script.

The subprogram specification is similar to an Ada subprogram specification [LRM, 6.1]. It may specify either a procedure or a function. Parameter declarations are separated by semicolons. Parameters and function return values must be of one of MCL's pre-defined types (Sec. B.2). Each formal parameter name must follow the format of a MCP variable. An example of a subprogram simulation statement is:

```
Procedure COMPILE_AND_LIST (%FILE:STRING;
                            %LIB: STRING;
                            %COMPILE_STATUS: out STRING)

IS
  BEGIN
   %DEBUG:= "LOOP READ_LINE %C; EXEC %C;END LOOP"
   COMPILE %FILE LIB=>%LIB
   %COMPILE_STATUS := %STATUS.EXIT_STATUS
   EXEC %DEBUG -- reads and executes user
               -- commands until an EXIT command.
               -- allows the interactive user to
               -- examine script variables, etc.
   IF %COMPILE_STATUS = OK THEN LPT %FILE
  END
```

If this text were placed in a database object named COMPLIST, the MCP user could invoke it by typing

```
: COMPLIST MYFILE MYLIB
```

Treatment of parameters is identical to that for program invocation. The script receives parameters and performs type checking based on the parameter specification. If an actual parameter is not of the proper type, and cannot be implicitely converted to the proper type, the script's execution is terminated. Otherwise, the script performs the commands in its body, and returns updated values of OUT parameters. If a script describes a function, the function return value must be specifed in a RETURN command. As in program invocation, default OUT parameters are generated on behalf of the user. In the example above, a variable named %COMPILE_ERRORS would be generated.

A.16.2 Affecting the MCP's Environment

A script is functionally equivalent to a program, and cannot directly modify the invoking MCP's environment. For example, any variables declared in a script cannot be subsequently referenced in the invoking MCP. The user may specify that the current MCP should

execute a sequence of MCL commands contained in a database object via the EXEC command. For example, if a database object named "VARIABLES" contained the text:

```
%MYLIB := ADALIB.YOURLIB.MYLIB

%MYFILE := ADALIB.SYSLIB.TESTFILE
```

the user might type

```
: EXEC CONTENTS(VARIABLES)

: -- variables defined in VARIABLES are now visible

: -- in the current environment

%MYLIB

  ADALIB.YOURLIB.MYLIB
```

APPENDIX B: MCL LANGUAGE ELEMENTS

B.1 Lexical Elements

(a) Character Set. Identical to Ada [LRM, 2.1]. This includes a basic graphic character set, as well as other characters from the ASCII graphics set. Any command can be expressed using only the basic character set. Any lower-case letter is equivalent to the corresponding upper-case letter except within character strings and unquoted strings.

(b) Lexical Units. An MCL input stream is a sequence of lexical units. The lexical units are identifiers (including reserved words), numeric literals, character literals, strings, delimiters, and comments.

A delimiter is either one of the following special characters in the basic character set:

```
&'()*+,-./:;<=>"
```

one of the following compound symbols:

```
=> ..  :=  /= >= <= -> -< -| -& --
```

or the character:

```
?
```

from the ASCII graphics set. Adjacent lexical units may be separated by spaces or by passage to a new line. An identifier or numeric literal must be separated in this way from an adjacent identifier or numeric literal. Spaces must not occur within lexical units, except within strings and comments. Each lexical unit must fit on one line. [LRM, Sec. 2.2]. A backslash ' ' character immediately preceding the newline character allows line continuation. The character preceding the backslash is catenated to the next line without any intervening characters.

(c) Identifiers. MCL identifiers are identical to Ada identifiers. As in Ada, identifiers differing only in the use of corresponding upper and lower case letters are considered to be the same. Examples:

```
INT1    Ada_LRM
```

(d) Numeric Literals. There are two classes of numeric literals: integer literals and real literals. Integer literals are the literals of the MCL type INTEGER. Real literals are the literals of the MCL type REAL. As in Ada, isolated underscore characters may be inserted between adjacent digits of a decimal number, but are not significant. [LRM, 2.4]. The conventional decimal notation is used. Real literals are distinguished by the presence of a decimal point.

An exponent indicates the power of 10 by which the preceding number is to be multiplied to obtain the value represented. An integer literal can have an exponent; the exponent must be positive or zero.

Examples:

```
12 0 123_4561E6 -- integer literals

12.0  0.0  0.456  3.14159_26  -- real literals

12.34E-4   -- real literal with exponent
```

(e) Character Strings. Identical to Ada [LRM, 2.6].

Examples:

```
""      -- empty string

"ABC"

""""    -- a single included string bracket character
```

If a character string conforms to the Ada identifier syntax and does not conflict with any reserved or predefined keywords, the string bracket characters (") may be omitted.

(f) Aggregates. An aggregate is a sequence of component values. As in Ada, components may be named or positional. [LRM, 4.3] Unlike Ada, only one choice may be specified for each component value and the choice must be an identified string.

(g) Database Literals. Database literals name database objects. There are two classes of database literals: partition specifiers and object names (see AIE(1).KAPSE(1)). The user may specify database literals using full attribute value notation or shorthand positional notation.

The first character of a database literal indicates the desired window. If the first character is a "." or "'", the database literal is rooted in the MCP's context object. Otherwise, the database literal is rooted in the CURRENT_DATA window associated with each user via the KAPSE.

A database literal beginning with a "." or "'" has different meanings to different program invocations, since each invocation has its own context object. However, each invoked program does automatically receive a window on the context object of its invoker, its invoker's invoker, and so on, back to the first program invoked by the system initiator. Therefore, a user-typed database literal beginning with a "." automatically has the name of the window on the MCP's context object appended to it by the MCP. This process, referred to as normalizing the database literal, allows the literal to be passed to an arbitrary invoked program without changing its meaning.

The MCP's context object is automatically deleted at the conclusion of the MCP session. Any database object whose name begins with a "." or "'" is thus a temporary object that will be deleted along with the context object. For example:

```
TEST            -- a database object named "TEST"
                -- located in the CURRENT_DATA window

.TEST           -- a temporary database object named "TEST"
                -- If the window on the MCP's context
                -- object were named
                -- "F4", this database literal would be
                -- normalized into  ".F4.TEST"
FLIGHT.SIM.*    -- a partition

FLIGHT.(RELEASE=>3,MODULE=>STICK)  -- attribute value list
                                   -- notation
```

(h) Comments. Identical to Ada.

Examples:

```
-- comments have no effect on the meaning of commands;
-- their sole purpose is the enlightenment of the human
-- reader
-- [LRM, 2.7]
```

(i) Keywords. The identifiers below are called reserved words and are reserved for special significance in the language.

ABORT AND ARRAY BEGIN CASE ELSE ELSIF END EXIT FOR FUNCTION IF IN LOOP MOD NOT OR OUT PROCEDURE RECORD REM RENAMES RETURN REVERSE USE THEN WHEN WHILE XOR LOGOUT TYPE

The following identifiers are predefined and have special significance in the language.

GET PUT SUSPEND RESUME WAIT HELP EXEC STOP START CANCEL STATUS DUMP_VARS INTEGER REAL STRING BOOLEAN

B.2 Types

MCL contains no facility for declaring types. A MCP literal must be one of the following predefined types:

1. INTEGER - This predefined type is implemented as in Ada [LRM, 3.5.4]. INTEGER values are expressed as integer literals.

2. REAL - This predefined type describes floating point real numbers. Its Ada declaration is

   TYPE REAL is digits NUM_DIGITS

where NUM_DIGITS is within the range of the most accurate numeric type supported by the implementation. REAL values are expressed as real literals.

3. STRING - The predefined type STRING denotes an unconstrained, one-dimensional array of characters. Its Ada declaration is

```
TYPE STRING is array (NATURAL range<>) of CHARACTER;
```

   STRING values are expressed as character strings. Catenation is a predefined operator for strings; it is represented as &. The relational operators <, <=, >, and >= are defined for strings, and correspond to lexicographical order.

4. BOOLEAN - As in Ada, there is a predefined enumeration type named BOOLEAN [LRM, 3.5.3]. It contains the two literals FALSE and TRUE ordered with the relation FALSE<TRUE.

5. AGGREGATE - The predefined types ARRAY and RECORD are expressed as aggregate literals. Component association for array types is strictly positional. RECORD component association may be both named and positional. The component values may each evaluate to any MCL type. Component selection and catenation are defined for aggregates.

B.3 Expressions

MCL expressions are a subset of Ada expressions. See Appendix C for MCL expression syntax. Note that a carriage return cannot appear between an operand and an operator of an expression, since a single operand is itself a valid expression.

Examples of expressions:

```
HELLO                   -- unquoted string

HELLO & GOODBYE         -- HELLOGOODBYE

%Q OR (%V AND FALSE)    -- variable names begin with a '%'

2 * 3. + 5              -- 11.
```

B.3.1 Operands

An operand of an expression may be a literal, a function call, a variable or a database attribute.

A literal denotes an explicit value of any legal MCL type. Examples of literals:

```
4            -- integer literal
4.0          -- real literal
TRUE         -- Boolean literal
"OFF"        -- string literal
OFF          -- unquoted string
(X,Y)        -- ARRAY literal
(A=>X,Y)     -- RECORD literal
```

A function call returns a value which can be used as an operand.

B.3.2 Operators

MCL operators, a subset of Ada, are divided into five classes. They are given in Table B.1 in order of increasing precedence.

B.3.3 Type Conversions

Any unquoted strings within an expression are implicitly converted to strings. In the following discussion , the term STRING refers to both strings and implicitly converted unquoted strings.

Each of the operations described expects its operands to be of specific types. If an operand is not of the proper type, the MCP attempts to implicitly convert it into a value of the proper type. If the conversion fails, an error message is printed. This conversion only occurs between closely related simple types, as shown in Table B.2.

For operators whose operands can be one of several different types, implicit conversion is attempted in the following order: BOOLEAN, REAL, INTEGER, STRING. This ordering implies that: (1) relational operators and catenation can be applied to operands of any type, since any type can be implicitly converted to a string; and (2) the result of adding, subtracting, multiplying or dividing an integer and a real is a REAL. Examples of implicit conversion:

```
FLIGHT & SIM          -- "FLIGHTSIM"
TRUE AND "FALSE"      -- FALSE
3+"5"                 -- 8
2.3 + 3               -- 5.3
"HELLO" & 2.0         -- "HELLO2.0"
```

TABLE B.1: MCL OPERATORS

Operator	Operation	Operand Type	Result Type
(Logical)			
AND	conjunction	BOOLEAN	BOOLEAN
OR	inclusive disjunction	BOOLEAN	BOOLEAN
XOR	exclusive disjunction	BOOLEAN	BOOLEAN
(Relational)			
= /=	equality and inequality	Any MCL Type	BOOLEAN
< <= > >=	test for ordering	Any MCL type	BOOLEAN
(Equality for reals is determined as in Ada [LRM 4.5.8])			
(Adding)			
+	addition	Integer or real	same type
-	subtraction	Integer or real	same type
&	catenation	Any MCL Type	string or aggregate
(Unary)			
+	identity	Integer or real	same type
-	negation	Integer or real	same type
NOT	logical negation	BOOLEAN	BOOLEAN
(Multiply-ing)			
*	multiplication	Integer or real	same type
/	division	Integer or real	same type
**	exponentiation	Integer or real	same type
MOD	modulus	Integer	Integer
REM	remainder	Integer	Integer

B.3.4 Explicit Type Conversions

Explicit functions exist to convert arguments from any given type to any given type. They are overloaded, that is, they take arguments of any "closely related type" (see Table B.2), and convert to the type named by the function. They are BOOLEAN, REAL, INTEGER, and STRING.

Table B.2: IMPLICIT TYPE CONVERSIONS

Proper Operand Type	Closely Related Types
BOOLEAN	STRING(value must be "TRUE" or "FALSE")
REAL	STRING(value must be the image of a real number) INTEGER
INTEGER	STRING(value must be the image of an integer) REAL (rounded to integer)
STRING	BOOLEAN REAL INTEGER

PAGE LEFT E

APPENDIX C: BNF FOR MAPSE COMMAND LANGUAGE (MCL)

--LEXICAL ELEMENTS

```
upper_case_letter ::= A | B | C | D | E | F | G | H | I |
                      J | K | L | M | N | O | P | Q | R | S
                      | T | U | V | W | X | Y | Z

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

special_characters ::= " | # | % | & | ( | ) | star | + |
               comma | - | dot | / | colon | semicolon |
               < | = | > | underscore | parallel_bar | tic

star ::= <*>

comma ::= <,>

dot ::= <.>

colon ::= <:>

semicolon ::= <;>

underscore ::= <_>

parallel_bar ::= <|>

blank ::=  < >

tic ::= <'>
```

```
lower_case_letter ::= a | b | c | d | e | f | g | h | i | j
                    | k | l | m | n | o | p | q | r | s | t | u
                    | v | w | x | y | z

other_special_characters ::=
 ! | $ | question_mark | @ | [ | / | ] | ^ | { | } | ~

question_mark ::= <?>

identifier ::= letter{[underscore]letter_or_digit}
  -- identical to ADA

letter_or_digit ::= letter | digit

letter ::= upper_case_letter | lower_case_letter

numeric_literal ::= decimal_number

decimal_number ::= integer [.integer][exponent]

integer ::= digit{[underscore]digit}

exponent ::= E [+] integer | E - integer

character_string ::= "{character}"

character ::= letter_or_digit | special_characters | blank |
       other_special_characters

string ::= database_literal | character_string

database_literal ::= objname | partition

objname ::= [path_sep] attribute { path_sep attribute }

path_sep ::= dot | '

partition ::= [path_sep] attribute_or_stare | { path_sep attribute_
      or_star }

attribute ::= identifier | aggregate

attribute_or_star ::= attribute | star

aggregate ::= ( component_association |
      { sep component_association } )

component_association ::= [ choice => ] expression

choice :: = identifier | identifier tic identifier -- the latter
      -- is for database literals only
-------------------------------------------------------------------
```

```
     -- EXPR PROC

expression ::= limited_expression | database_literal
 -- A single database_literal may be an expression,
 -- except within an expression statement

limited_expression::= limited_relation { logical_operator relation}
          | expression { logical_operator relation}

relation ::= simple_expression
          | relation relational_operator simple_expression

limited_relation ::= limited_simple_expression
          | relation relational_operator simple_expression

simple_expression ::= [unary ] term
          | simple_expression adding_operator term

limited_expression ::= unary term
          | limited_term
          | simple_expression adding_operator term

term ::= primary
          | term multiplying_operator primary

limited_term ::= limited_primary
          | term multiplying_operator primary

primary ::= database_literal
          | limited_primary

limited_primary ::= literal
          | variable_expression
          | ( expression )

literal ::= character_string
          | decimal_literal

logical_operator ::= AND | OR | XOR

relational_operator ::= = | /= | < | <= | > | >=

unary_operator ::= + | - | NOT

adding operator + | - | &

multiplying_operator ::= * | / | ** | MOD | REM

variable_expression ::= %identifier { component_selector }

component_selector ::= ( expression ) -- Positional selection
          | dot identifier  -- named Selection.
```


INTERMETRICS INCORPORATED • 733 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

```
--COMMANDS

command_input ::= command {terminator command}

terminator ::= newline | semicolon


newline ::= [comment] <CR>

comment ::= -- { character }
 -- As in ADA

command ::=

[label] statement{-| statement } [-< objname ] [ -> objname ] [ -& ]
    | [label] statement {-1 statement} -> objname -< objname [-&]
 -- The notation -| between two commands indicates that
 -- the command are to be connected via a pipe.
 -- Standard output of the left side of a pipe becomes
 -- standard input of the right side of a pipe.
 -- The notation -& indicates that the command's execution should
 -- take place in the background. A task is generated to perform
 -- the command.
 -- The notation -< is used to indicate that the leftmost command's
 -- standard input should be directed from OBJNAME.
 -- The notation -> is used to indicate that the rightmost command's
 -- standard output should be directed to OBJNAME.
 -- Note that redirection may be specified in either order

label ::= identifier colon

statement ::= simple_statement | compound_statement


simple_statement ::= null_statement |
                program_or_script_call_statement |
                assignment_statement |
                return_statement  |
                suspend_statement  |
                logoff_statement  |
                abort_statement |
                wait_statement |
                dump_vars_statement|
                exit_statement |
                expression_statement |
                exec_statement |
                renames_statement |
```

```
compound_statement ::= if_statement | loop_statement
                     | script_statement
                     | block_statement

null_statement ::=

program_or_script_call_statement ::= objname[actual_parameter_part]
 -- OBJNAME may be a linked executable object or a MCP script.
 -- Invocation syntax is the same for either.

actual_parameter_part ::= parameter_association
                          {separator parameter_association}

separator ::= blank | comma
 -- Parameters are separated by blank or comma.
 -- However, if the user wishes to continue the parameter list
 -- on the next line, a comma is required.

parameter_association ::= [ formal_parameter => ] actual_parameter
 -- Named and positional parameters can be in any order in
 -- the parameter association. For example,
 --       prog  1  parm4 => 2  5
 -- The semantics of this ordering is as follows:
 -- All positional parameters are grouped together, followed
 -- by all named parameters.
 --
 -- Parameter values can be specified more than once in the
 -- same command invocation. For example,
 --       prog 1 parm4 => 2 5 parm4 => 3
 -- The MCP also implements default OUT parameters, as follows:
 -- If, at the end of parameter specification, an OUT
 -- parameter has been omitted, the MCP will implicitely
 -- declare a variable named %formal_parameter_name, and will
 -- generate a parameter specification of the form
 --       formal_parmameter => %formal_parameter_name
```

```
formal_parameter ::= identifier


actual_parameter ::= expression | help_mark


help_mark ::= question_mark
 -- Instead of typing an actual parameter value, the user can type
 -- a question mark to find out information about the parameter.
 -- Action taken depends on the "help" attributes of the
 -- invoked program or script.
 -- If those attributes indicate that the program
 -- wishes to handle help requests itself, the
 -- actual_parameter_part is considered to be complete, and the
 -- program is invoked.  The program will presumably take special
 -- actions when it encounters the '?' as an actual parameter
 -- value.
 -- If the "help" attributes indicate that help files are
 -- available, the MCP prints the appropriate
 -- file, and allows the user to continue specifying the
 -- actual_parameter_part for the program or script.


function_call ::= objname([actual_parameter_part])
 -- Unlike a program call, the function parameters
 -- must be surrounded by parentheses.
 -- As in a program call, help may be requested
 -- on a per-parameter basis.


assignment_statement ::= attribute := expression  |
                         variable := expression
      -- As in ADA, except that attribute references to database
      -- objects can appear on the statement's left hand side.


return_statement ::= RETURN [ expression ]
 -- Causes termination of the MCP. Control is returned to the
 -- MCP's invoker.
 -- If the MCP was processing a script which described a
 -- function, the function is given the return value described
 -- by the expression.
```

```
suspend_statement ::= SUSPEND objname
 -- Suspend the current MCP's execution and return control to
 -- the MCP's invoker.
 -- The MCP's state is saved in OBJNAME, which may be
 -- used as RESUME's argument.


logoff_statement ::= LOGOFF
 -- The current MCP session is terminated, along with the
 -- entire MAPSE session.


abort_statement ::= ABORT tasknames
 -- Abort the specified background tasks.


wait_statement ::= WAIT tasknames
 -- Wait for the specified background tasks to complete before
 -- accepting more commands.


tasknames ::= identifier {separator identifier}


expression_statement ::= limited_expression
 -- Any expression is legal but a database literal.


exec_statement ::= EXEC expression
 -- Executes the string described by the expression as a command.


if_statement ::= IF condition THEN command_input
               { ELSIF condition THEN command_input}
               [ ELSE command_input ] END IF
 -- Same as ADA


condition ::= expression
 -- Must evaluate to boolean.

loop_statement ::= [iteration_clause]  basic_loop
 -- Loops are legal in interactive as well as script mode.


iteration_clause ::= FOR variable IN discrete_range  |
                     WHILE condition
```

```
discrete_range ::= [REVERSE] expression [.. expression]

 -- If a single expression is specified, it must be an aggregate
    value.
 -- If both expressions are specified, they must be integer valued.

basic_loop ::= LOOP command_input END LOOP

exit_statement ::= EXIT

block_statement ::= BEGIN command_input END


script_statement ::=
          script_header IS block_statement [identifier]
 -- Execution of this statement causes the MCP to import
 -- from the MCP's context object
 -- parameters specified in the SUBPROGRAM_SPECIFICATION,
 -- execute the commands contained in BLOCK_STATEMENT,
 -- then write updated OUT parameter values back to
 -- the MCP's context object.
 -- This enables the user to create a MCP script which simulates
 -- an ADA program.


script_header ::=
 PROCEDURE identifier [formal_part]      |
 FUNCTION  identifier [formal_part] RETURN subtype_indication
    -- General scheme for script parameters: their formal name
    -- must have the same format as any MCP variable - i.e.,
    -- %identifier. Thus, they can be treated as any MCP
    -- variable.
    -- The program invoking the MCP script may optionally
    -- omit the leading '%' when specifying a script's
    -- formal parameter name.


formal_part ::=
( parameter_declaration { terminator parameter_declaration }  )
 -- Parameter declarations can be separated by newline or semicolon.


parameter_declaration ::=
mcp_identifier_list: mode subtype_indication [:= expression]


mcp_identifier_list ::= variable { , variable }

mode ::= [IN] | OUT | IN OUT


subtype_indication ::= INTEGER | BOOLEAN | STRING | REAL | ARRAY
                     | RECORD
 -- Any legal MCL type.

dump_vars_statement ::= DUMP_VARS
```

APPENDIX D: STATEMENTS AND FUNDAMENTAL PROGRAMS

The following tables summarize those MCL commands which are statements (i.e., executed via internal MCP actions) and those which are effected via invocation of a fundamental program.

TABLE D.1: STATEMENTS

Name	Reason for Internal Execution
get put dump_vars	References internal MCP variables.
return logoff suspend	Checks for active internal background tasks, terminate MCP invocation.
wait abort	Affects the status of background tasks.
if loop exit	Modifies flow of MCL command execution.
script header	Reads and writes parameters from MCP's context object.
block	Notation following the block (I/O redirection, pipe or background) must be interpreted by the MCP.
exec	String is evaluated by the MCP as if it were typed by the user as a command
renames	Modifies internal symbol table

TABLE D.2: FUNDAMENTAL PROGRAMS

Name	Comments
HELP	
RESUME	Invokes the MCP. RESUME's argument specifies the MCP's context object.
STOP	KAPSE function.
START	"
CANCEL	"
STATUS	"
database manipulation commands	"

TABLE D.3: COMMANDS. ADA - MCL COMPARISON

MCL Command	Ada Statement	Differences
Program Call	Procedure Call	1. program name replaces the procedure name. 2. parentheses surrounding the actual parameter part are omitted. 3. parameter associations may be separated by a blank. 4. positional and named parameters may be freely mixed. 5. default OUT parameters are generated. 6. parameter help available.
assignment	assignment	left-hand side may describe a database attribute.
get	get	1. reads from standard input only. 2. reads text only. 3. reads a variable number of values.
put	put	1. writes to standard output only. 2. writes text only. 3. writes a variable number of values.

TABLE D.3: COMMANDS (Cont'd.)

MCL Command	Ada Statement	Differences
if	if	—
loop	loop	1. for loop may generate over the component fields in an aggregate.
exit	exit	1. only the enclosing loop may be exited. 2. no condition may be associated with the exit.
return	return	—
abort	abort	—
block	block	—
script header	subprogram specification	formal parameter names must follow the format of MCP variables (i.e., must begin with '%').

TABLE D.4: LANGUAGE ELEMENTS. ADA - MCL COMPARISON

MCL Element	Ada Element	Differences
numeric literals	numeric literals	———
string literals	string literal	may be unquoted.
boolean literals	boolean literals	———
types	types	pre-defined set.
variables	variables	name must be preceded by '%'.
expressions	expressions	no and then, or else operators.

END

FILMED

11-83

DTIC